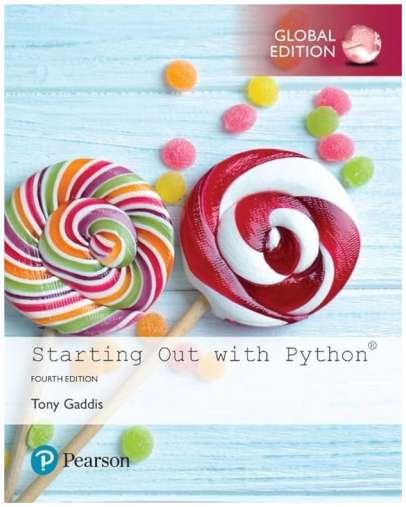
Lists, Strings and Tuples

CIIC 3015 - Lecture 5

Jean Carlos Méndez

Lecture Goals

- **Lists**
  - What are they?
  - Why do we need them?
  - How to create and use them?
- **Strings**
  - String functions!
  - Differences with List
- **Tuples**
  - What are they?
  - How to create and use them?
- **2D Lists**
  - What are they?
  - Why do we need them?
  - How to create and use them?



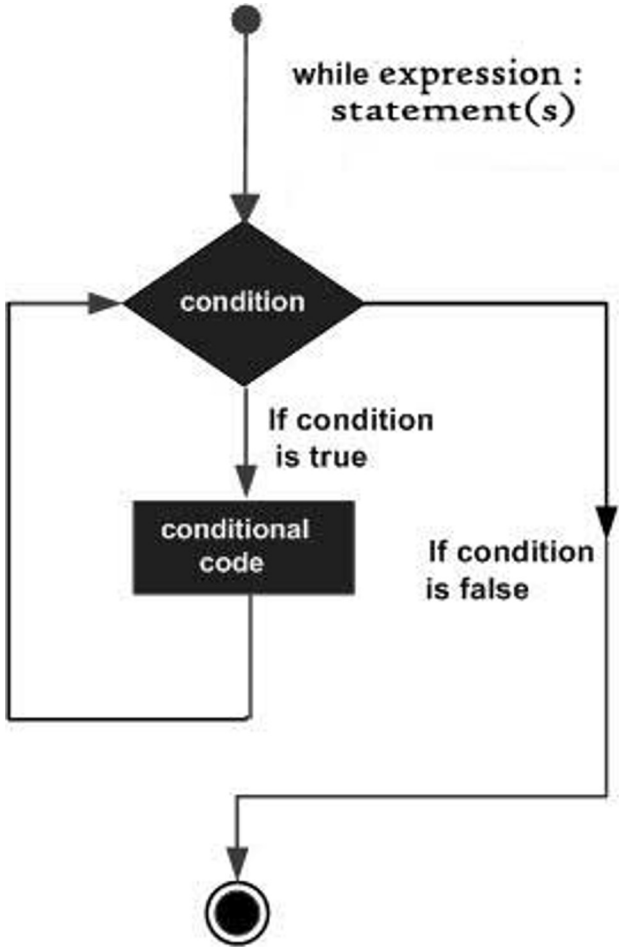
Starting out with Python
(4th Edition)
Chapters 7 and 8

Quick Recap

Recap: **while** loops

Control structure that runs a block of code *repeatedly* until a certain condition is no longer met

```
while condition is true:

  do something
```



Recap: for loops

- Control structure that runs block of code repeatedly by iterating over elements in sequence

```
for letter in "Colegio":
  print(letter)
```

Recap: **while** vs. **for** loops

Print the numbers from 0 to 9:

```
index = 0
while index < 10:
    print(index)
    index = index + 1
print('All done!')
```

```
for i in range(0, 10):
  print(i)
print('All done!')
```

Recap: Sequences

- An ordered collection of items that can be iterated over.
- So far we have seen **strings**, which are sequences of characters.
- Python gives us a handy **range()** function to create a sequence of integers.
- We will talk more about sequences in our next lecture, but for now let's just think of these two types of sequences for now.

Google | GH |

Lists

Recall: Data Types

int	Number with **no** decimal point	`5, -17, 0, 2874976`
float	Number with decimal point	`2.5, 3.00, -12.95, 7.`
str	Text between quotes	`'Hello', "", "False", '53.12'`
bool	True or False	`True, False`

In Python these are the four **Primitive Data Types**, because they are the most basic.

Google | GH |

Lists: Introduction

- We will now expand our list of data types with our first non-primitive data type: **Lists**.
- Lists are different than the primitive data types we have seen because they are a **collection** of the types we have seen.
- Example:

Example of an **int**... **5**

Example of a **List** of **int**s... [**5**, **6**, **7**]

Lists: Introduction

Example of an **int**…**5**	Example of a **List** of **int**s…[**5**, **6**, **7**]
Example of a **float**…**5.0**	Example of a **List** of **float**s…[**5.0**, **6.0, 7.0**]
Example of a **str**…**"ok"**	Example of a **List** of **str**s… [**"ok", "haha"**]
Example of a **bool**…**True**	Example of a **List** of **bool**s… [**True, False**]

- But why would we need to do this?
- **Lists** help us perform operations on **large amounts of data** in a much **more efficient** manner than just working on each data point individually.
- Let's look at a real-world example to see why this is the case…

Google | GH |

Lists: Motivating Example

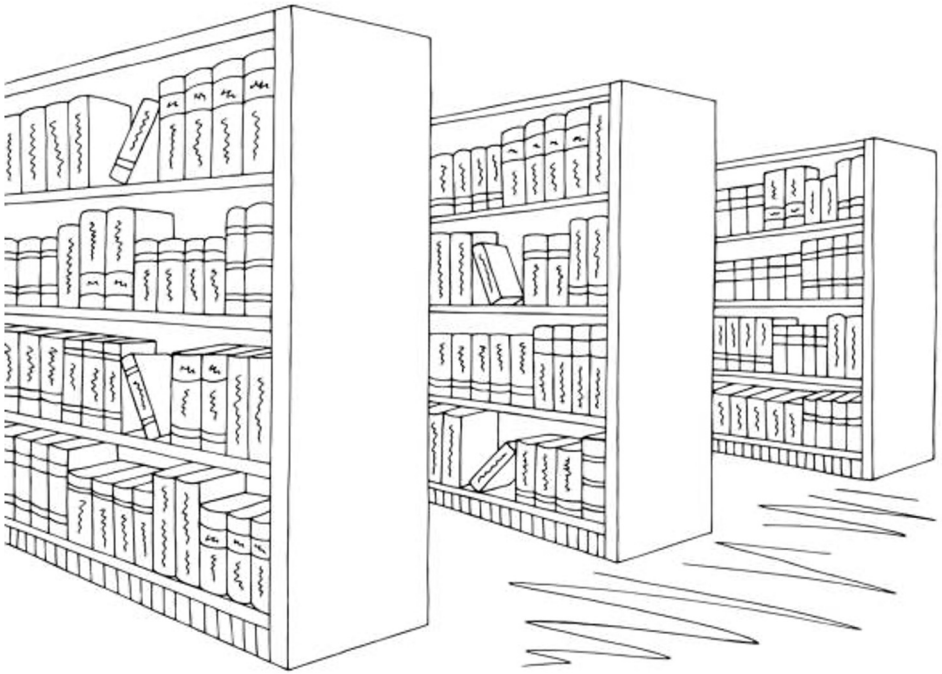
- Imagine you put books in a pile as people return them to the library…

Lists: Motivating Example

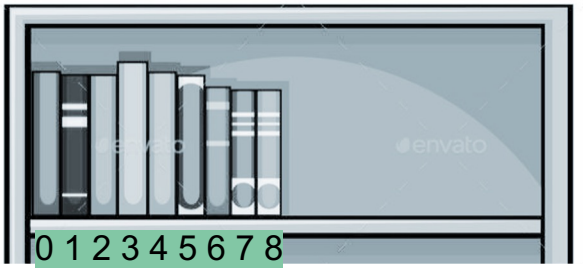
- Eventually, finding a book in a huge pile becomes too much work…

Lists: Motivating Example

- So real libraries use **numbered** shelves to **organize** stored books.



Good strategy: storing books in **shelves**



The book you want is on that shelf at position (a.k.a. index) [6]

Can you imagine a world without lists?

main.py ×

```
1  # return student N
2  student1 = 'Alanna'
3  student2 = 'Brian'
4
5  studentNumber = int(input("What student do you want? "))
6  if (studentNumber == 1):
7    print(student1)
8  elif (studentNumber == 2):
9    print(student2)
10
```

Console Shell

```
What student do you want? 2
Brian
>
```

Can you imagine a world without lists? Not Pretty!

- What if there's **MANY** students? This quickly becomes too much work…

main.py

```
# return student N
student1 = 'Alanna'
student2 = 'Brian'
student3 = 'Chet'
student4 = 'Darian'
student5 = 'Eduardo'
student6 = 'Fred'
student7 = 'Gretchen'
student8 = 'Harper'
student9 = 'Ivy'
student10 = 'Luke'
student11 = 'Madison'
student12 = 'Nora'
student13 = 'Oliver'
student14 = 'Phoebe'
student15 = 'Quest'
student16 = 'Ruby'
student17 = 'Steve'
student18 = 'Taylor'
student19 = 'Uma'
student20 = 'Violet'

studentNumber = int(input("What student do you want? "))
if (studentNumber == 1):
  print(student1)
elif (studentNumber == 2):
  print(student2)
elif (studentNumber == 3):
  print(student3)
elif (studentNumber == 4):
  print(student4)
elif (studentNumber == 5):
  print(student5)
elif (studentNumber == 6):
  print(student1)
elif (studentNumber == 7):
  print(student2)
elif (studentNumber == 8):
  print(student3)
elif (studentNumber == 9):
  print(student4)
elif (studentNumber == 10):
  print(student5)
elif (studentNumber == 11):
  print(student11)
elif (studentNumber == 12):
  print(student12)
elif (studentNumber == 13):
  print(student13)
elif (studentNumber == 14):
  print(student14)
elif (studentNumber == 15):
  print(student15)
elif (studentNumber == 16):
  print(student16)
elif (studentNumber == 17):
  print(student17)
elif (studentNumber == 18):
  print(student18)
elif (studentNumber == 19):
  print(student19)
elif (studentNumber == 20):
  print(student20)

```

plus, what happens if a new student "Bob" joins/leaves the class?

Rename all our variables?!?

Console Shell

```
What student do you want? 18
Taylor
```

Lists: Motivating Example

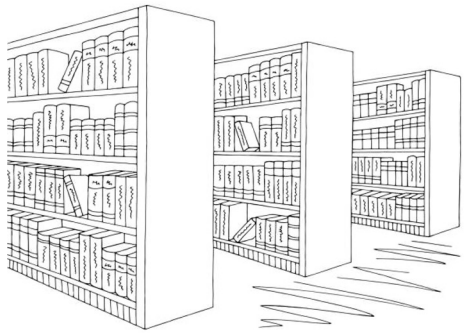
- Lists make all this much easier and efficient!

main.py

```
# return student N
students = ['','Alanna','Brian','Chet','Darian','Eduardo','Fred','Gretchen','Harper','Ivy','Luke','Madison','Nora','Oliver','Phoebe','Quest','Ruby','Steve','Taylor','Uma','Violet']

studentNumber = int(input("What student do you want? "))
print(students[studentNumber])

```

Console Shell

```
What student do you want? 18
Taylor
```



Google

Lists: Definition

- **Data structure** containing **ordered list of elements**, which can be **strings**, **numbers**, **other data types**, and other data structures (including other lists).
- It is a data type, just like **str**, **int**, **bool**, **float**
- Can **index**, **slice**, **in**, **iterate**
- Holds data that needs to stay in a **certain order**.

Lists: Syntax

- **Create list**: putting comma-separated values inside of square brackets **[]**
- Can contain **zero or more values of any type**
  - Usually same type in one list
- Can contain multiple types, but less typical
- Examples:

```
ages = [18, 26, 19, 21, 23, 34]
states = ["AL", "MS", "SC", "CA"]
answers = [True, True, False, True]
weird = [True, 1, 1.0, '1']
```

Lists: Indexing

```
                0    1    2    3    4    5
my_list = [18, 26, 19, 21, 23, 34]
```

`my_list[0]`	**18**
`len(my_list)`	**6**
`my_list[4]`	**23**
`my_list[5]`	**34**
`my_list[len(my_list)]`	**IndexError**

Google | GH |

Lists: Negative Indexing

```
                0    1    2    3    4    5
my_list = [18, 26, 19, 21, 23, 34]
```

```
len(my_list)                  6

my_list[len(my_list)-1]       34  # Last element

my_list[len(my_list)-2]       23  # Second to last element

my_list[-1]                   34  # Last element

my_list[-2]                   23  # Second to last element
```

Negative indexing lets you get the last elements of the list without as much typing!

Google | GH |

Lists: Slicing and **in**

```
                0    1    2    3    4    5
my_list = [18, 26, 19, 21, 23, 34]
```

`my_list[1:4]`	**[26, 19, 21]**
`21 in my_list`	**True**
`my_list[:3]`	**[18, 26, 19]**
`100 in my_list`	**False**
`my_list[4:]`	**[23, 34]**
`my_list[3:len(my_list)-1]`	**[21, 23]**

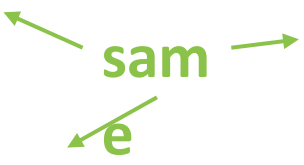
Google

Lists: Iteration

```
my_list = ['abc', 'xyz', 'hello', 'goodbye']
```

```
for item in my_list:
    print(item)
```

```
for i in range(len(my_list)):
    print(my_list[i])
```




```
i = 0
while i < len(my_list):
    print(my_list[i])
    i = i + 1
```

Lists: Tracing for loops

```
for num in [1, 20, 3]:
    print(num)
print('no more numbers')
```

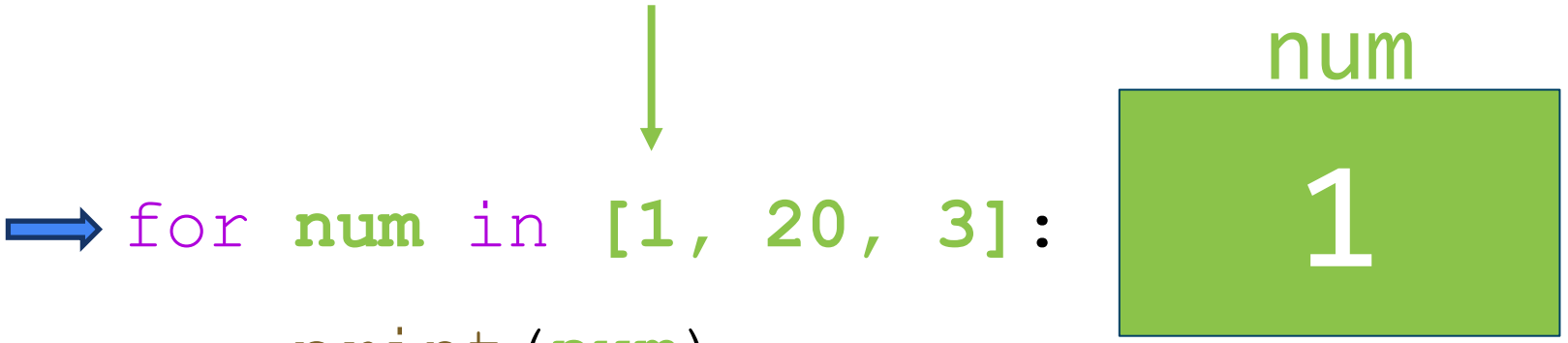
num

Google | GH |

Lists: Tracing for loops

```
for num in [1, 20, 3]:
    print(num)
print('no more numbers')
```

Lists: Tracing for loops

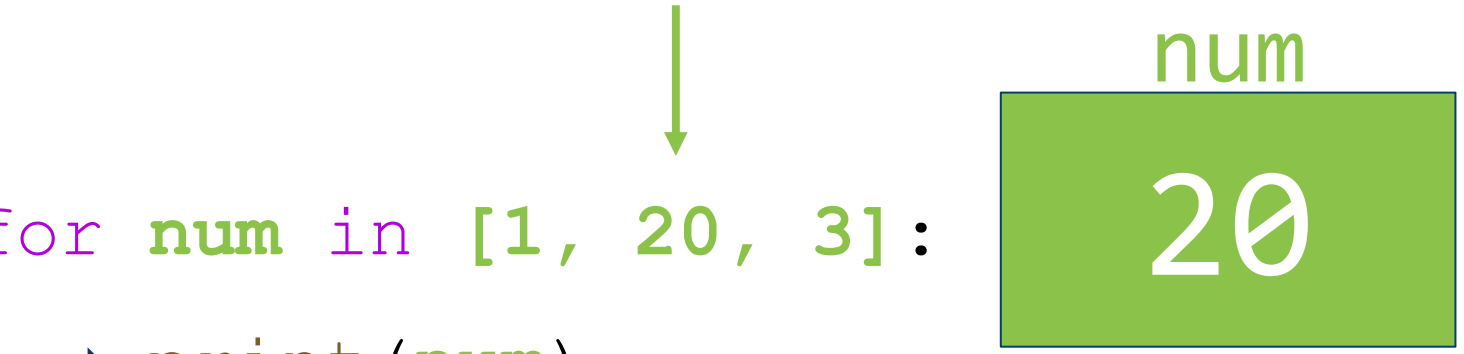



```
for num in [1, 20, 3]:
    print(num)
print('no more numbers')
```

```
1
```

Google | GH |

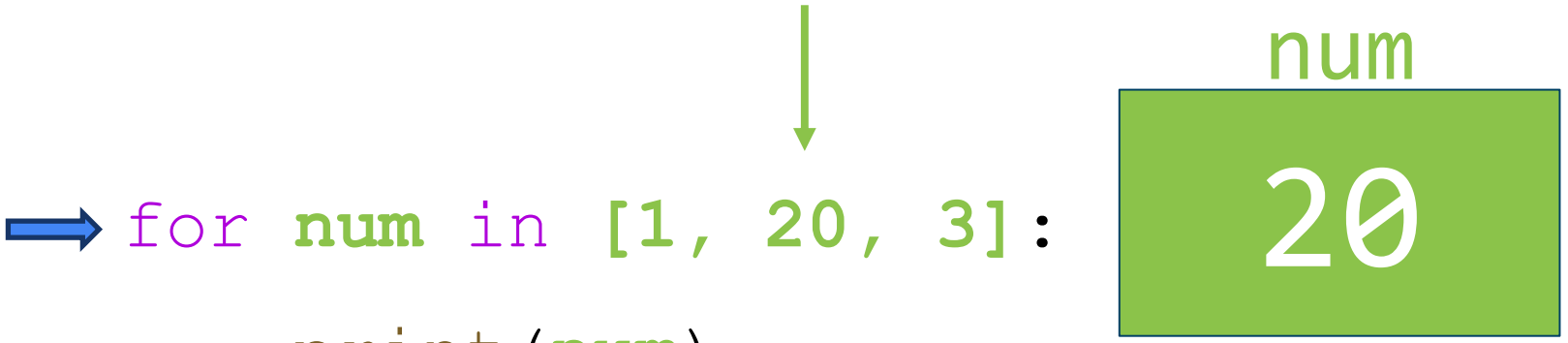
Lists: Tracing for loops




```
for num in [1, 20, 3]:
    print(num)
print('no more numbers')
```

```
1
```

Lists: Tracing for loops




```
for num in [1, 20, 3]:
    print(num)
print('no more numbers')
```

```
1
20
```

Google | GH |

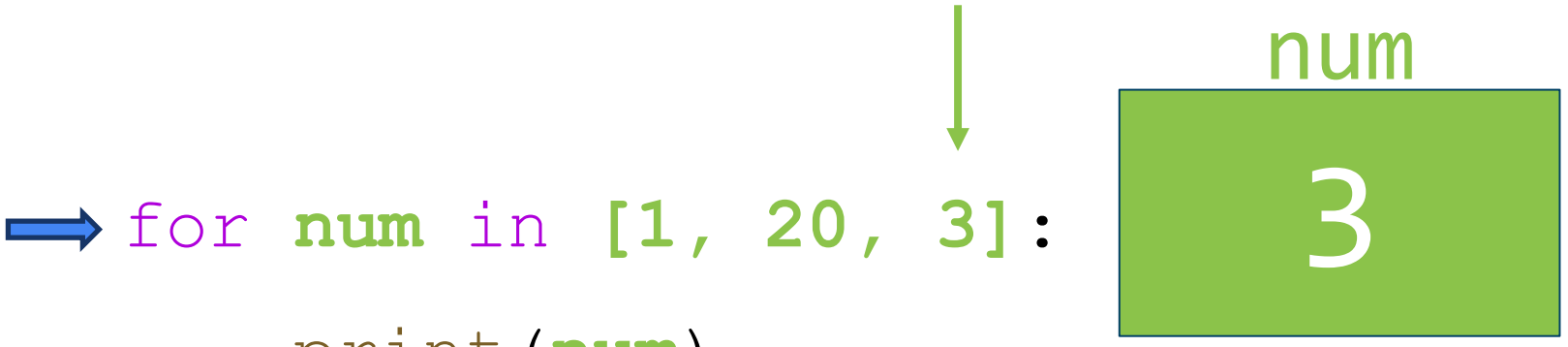
Lists: Tracing for loops

```
for num in [1, 20, 3]:
    print(num)
print('no more numbers')
```

```
1
20
```

Google | GH |

Lists: Tracing for loops

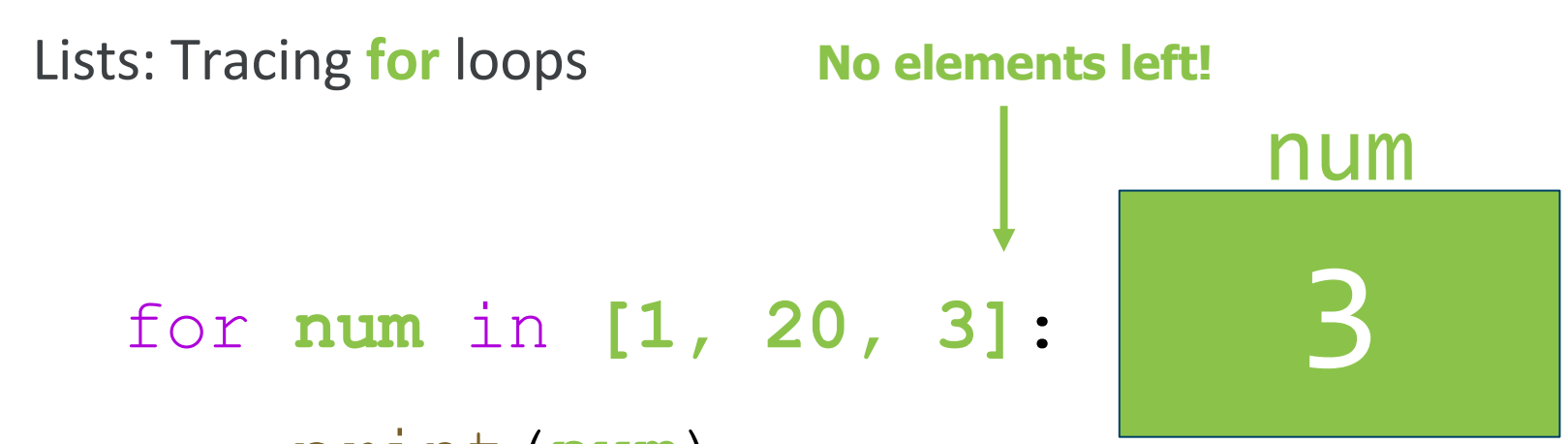



```
for num in [1, 20, 3]:
    print(num)
print('no more numbers')
```

```
1
20
3
```

Google | GH |

Lists: Tracing for loops



Lists: Tracing for loops

```
for num in [1, 20, 3]:
    print(num)
print('no more numbers')
```

num

```
1
20
3
no more numbers
```

Google | GH |

Live Coding! Example 1

- Create Lists
- Iterate over the lists (using both **while** and **for** loops).

Code available in:

github.com/CSwithJC/CIIC3015/blob/main/lecture_05/example_01.py

Lists: Mutability

- **mutable**: once created, value can change
- **immutable**: once created, value cannot change

- **immutable** types:
  - **int**
  - **float**
  - **bool**
  - **str**

mutable types:

- list
- …?

Lists: Mutability

- Lists are **mutable**: we can change their elements
- Strings are **immutable**: we cannot change their characters

Mutability of Variables vs. Values

- But what about variables? Haven't we been changing their values all semester?

Assigning a new value to a variable is **not** mutating a value

```
x = 10
x = x + 1
x = 'Hello'
x = x[1:4]
```

That's just *replacing* value held by the variable

Google | GH |

Lists Mutation: **Change an Item**

To update a single value in a list, you assign to the *indexed* value.

```
grades = [95, 70, 92]
print(grades)  # prints "[95, 70, 92]"
grades[1] = 90
print(grades)  # prints "[95, 90, 92]"
```

To update multiple values at once, you can assign to the *sliced* value.

```
grades = [95, 70, 92]
print(grades)  # prints "[95, 70, 92]"
grades[1:] = [90, 100]
print(grades)  # prints "[95, 90, 100]"
```

Lists Mutation: **Change an Item**

Strings on the other hand, are **immutable**. Trying to modify an element after its created yields an TypeError!

```
name = "Andre"
name[0] = "a" # TypeError!
```

Lists Mutation: **Change an Item**

```
friends = ['Joey', 'Rachel', 'Ross', 'Phoebe', 'Monica']
friends[1] = 'Sabrina'                  ['Joey', 'Sabrina', 'Ross', 'Phoebe', 'Monica']
friends[3] = 'James'                    ['Joey', 'Sabrina', 'Ross', 'James', 'Monica']
friends[0] = friends[2]                 ['Ross', 'Sabrina', 'Ross', 'James', 'Monica']
friends[len(friends) - 1] = 'Serena'    ['Ross', 'Sabrina', 'Ross', 'James', 'Serena']
friends[4] = friends[3] + ' is cool'    ['Ross', 'Sabrina', 'Ross', 'James', 'James is cool']
friends[2] = 24                         ['Ross', 'Sabrina', 24, 'James', 'James is cool']
friends = friends[3]                    'James'
```

Beware of List *Aliasing*

```
friends1 = ['Joey', 'Rachel', 'Ross', 'Phoebe', 'Monica']
friends2 = ['Joey', 'Rachel', 'Ross', 'Phoebe', 'Monica']
friends3 = friends1
friends1[3] = 'James'
print(f"friends1: {friends1}")
```

```
friends1: ['Joey', 'Rachel', 'Ross', 'James', 'Monica']
```

```
print(f"friends2: {friends2}")
```

```
friends2: ['Joey', 'Rachel', 'Ross', 'Phoebe', 'Monica']
```

```
print(f"friends3: {friends3}")
```

```
friends3: ['Joey', 'Rachel', 'Ross', 'James', 'Monica']
```

Lists Mutation? **Concatenate Lists**

`+` operator concatenates lists together

Note: This makes a new list from copies of the two given lists.

It does NOT mutate either of the original lists being concatenated.

Lists Concatenation

```
dogs = ['Cooper', 'Zoey']
cats = ['Magic', 'Finn']

dogs_and_cats = dogs + cats
```

['Cooper', 'Zoey', 'Magic', 'Finn']

```
pets = dogs + ['Shamu', 'Salem']
```

['Cooper', 'Zoey', 'Shamu', 'Salem']

Google | GH |

Lists Mutation Methods: **Add a single item**

`list.append(x)`

- Add element *x* to end of `list`

`list.insert(i, x)`

- Insert *x* item at a given position i in `list`
- First argument i is index of element before which to insert
- `a.insert(0, x)` inserts at front of list
- `a.insert(len(a), x)` is equivalent to `a.append(x)`

Lists Mutation: **Add an item**

```
cats = []                       cats = []

cats.append('Magic')            ['Magic']

cats.insert(0, 'Finn')          ['Finn', 'Magic']

cats.insert(1, 'Pixie')         ['Finn', 'Pixie', 'Magic']

cats.append('Rory')             ['Finn', 'Pixie', 'Magic', 'Rory']

cats.insert(3, 'Tigger')        ['Finn', 'Pixie', 'Magic', 'Tigger', 'Rory']
```

Lists Mutation: **Add many items to end**

`list1.extend(list2)`

- `list1` is a list
- Add `list1` to end of `list2`

Lists Mutation: **Add many items to end**

```
companies = []
companies.extend(['Google', 'Facebook'])
  ['Google', 'Facebook']
companies.extend(['Apple', 'Amazon', 'Zillow'])
  ['Google', 'Facebook', 'Apple', 'Amazon', 'Zillow']
companies.append(['Microsoft', 'Lyft'])
  ['Google', 'Facebook', 'Apple', 'Amazon', 'Zillow', ['Microsoft', ['Lyft']]]
```

Lists Mutation: **Delete an item**

`list.pop(i)`

- *i* is an optional index
- Remove item at index *i* from `list` and
- If no index is provided, remove from end of `list`
- Returns the removed item

`list.remove(x)`

- *x* is an item in `list`
- Remove first item from `list` whose value is equal to *x*
- If *x* does not exist in `list`, will raise `ValueError`
- `Returns None`

Lists Mutation: **Delete an item**

To remove an element from the list...

- Use `list_name.remove(item)` to remove an equal *item*
- Use `list_name.pop(index)` to remove and return an item at a *specific index*

```
items = [7, "Dindu", True]
items.pop(1)  # Removes and return "Dindu"
print(items)  # Prints [7, True]
items.remove(7)
print(items)  # Prints [True]
```

Lists Mutation: **Delete an item**

```
classes = ['Calculus', 'CS1', 'Theology']
classes.pop(1) # What does this return? What value is classes now?
     returns 'CS1'
     classes is now ['Calculus', 'Theology']

classes.remove('Theology')
     classes is now['Calculus']

classes.remove('Physics')
     ValueError

classes.pop()
     classes is now[]
```

Google | GH |

Lists Mutation: **Sorting a List**

`list.sort()`

- Sorts the list in increasing/alphabetical order

```
names = ["Jada", "Deb", "Pat"]

names.sort()

print(names)
```

```
['Deb', 'Jada', 'Pat']
```

Lists: **Slicing**

Slicing works for lists just like for strings (because lists and strings are both types of sequences, and slicing works for all sequences)!

```
tas = ["Deontae", "Jonathan", "Dindu", "Bradon"]

all_but_one_tas = tas[1:]
print(all_but_one_tas)  # ['Jonathan', 'Dindu', 'Bradon']
```

Slicing makes a (shallow) copy of the list.

Lists: **Rapid Fire Functions and Methods**

```
numbers = [5, 6, 12, 9, 4]

# Find largest number in list
max_num = max(numbers)  # == 12

# Find smallest number in list
min_num = min(numbers)  # == 4

# Find the sum of numbers in list
total = sum(numbers)  # == 36

fruits = ["apple", "banana", "orange"]
# Find the index of a specific value
banana_index = fruits.index("banana")  # == 1
```

Lists Mutation: **Accumulator Pattern**

Example: Create a list with values 0 through 4.

```
result = []
for num in range(0, 5):
    result.append(num)
print(result)
```

```
[0, 1, 2, 3, 4]
```

Live Coding! Example 2

- Add, remove and find items in Lists.
- Sort Lists.
- Create list with numbers 0 through 99

Code available in:
github.com/CSwithJC/CIIC3015/blob/main/lecture_05/example_02.py

Lists and Loops

We want to print **all** the elements in a **list**. What should our **condition** be?

```
shopping_list = ["Milk", "Eggs", "Cheese", "Bananas"]
index = 0
```

a. `while index < 3:`

b. `while index < len(shopping_list):`

c. `while index < len(shopping_list) - 1:`

d. `while index <= len(shopping_list):`

Lists and Loops

We want to print **all** the elements in a **list**. What should our **condition** be?

```
shopping_list = ["Milk", "Eggs", "Cheese", "Bananas"]
index = 0
```

a. `while index < 3:`

b. `while index < len(shopping_list):`

c. `while index < len(shopping_list) - 1:`

d. `while index <= len(shopping_list):`

Google | GH |

Lists and Loops

We want to print **all** the elements in a **list**. How can we print just one element of the list at a time?

```
shopping_list = ["Milk", "Eggs", "Cheese", "Bananas"]
index = 0
while index < len(shopping_list):
```

a. `print(shopping_list[index])`

b. `print(shopping_list)`

c. `print(index)`

Lists and Loops

We want to print **all** the elements in a **list**. How can we print just one element of the list at a time?

```
shopping_list = ["Milk", "Eggs", "Cheese", "Bananas"]
index = 0
while index < len(shopping_list):
```

a. `print(shopping_list[index])`

b. `print(shopping_list)`

c. `print(index)`

for Loop Patterns: Filters examples

Given a list of integers **nums**, create a function called **even** that returns a list with only the **even** numbers in **nums**.

- **Example**:
  - Input: `nums = [4, 1, 2, 3, 12]`
  - Output: `[4, 2, 12]`
- **Example**:
  - Input: `nums = [1, 3, 5, 7]`
  - Output: `[]`
- **Example**:
  - Input: `nums = []`
  - Output: `[]`

```
def even(nums):
  res = []
  for num in nums:
    if is_even(num):
      res.append(num)
  return res

def is_even(num):
  return num % 2 == 0
```

Google

for Loop Patterns: Filters examples

Given a list of integers **nums**, create a function called **positives** that returns a list with only the **positive** numbers in **nums**.

- **Example**:
  - Input: `nums = [4, -1, -2, -3, 12]`
  - Output: `[4, 12]`
- **Example**:
  - Input: `nums = [-1, -3, -5, -7]`
  - Output: `[]`
- **Example**:
  - Input: `nums = []`
  - Output: `[]`

```
def positives(nums):
  res = []
  for num in nums:
    if is_positive(num):
      res.append(num)
  return res

def is_positive(num):
  return num > 0
```

Google | GH |

for Loop Patterns: Mapping examples

Given a list of floats **nums**, create a function called **map** that returns a list with with all of the floats in **nums** turned into **strings**.

- **Example**:
  - Input: `nums = [4.0, -1, -2, -3.0, 12]`
  - Output: `['4.0', '-1', '-2', '-3.0', '12']`

```
def map(nums):
  res = []
  for num in nums:
    res.append(to_string(num))
  return res

def to_string(num):
  return str(num)
```

for Loop Patterns: Accumulation examples

Given a list of strings **my_list**, create a function called **concat** that returns a **string** with with all of the strings in **my_list** concatenated.

- **Example**:
  - Input: `my_list = ['hello', 'my', 'name', 'is', 'jc']`
  - Output: `'hellomynameisjc'`
- **Example**:
  - Input: `my_list = []`
  - Output: `''`

```
def concat(my_list):
  res = ''
  for num in my_list:
    res += str(num)
  return res
```

Google | GH |

Live Coding! Example 3

- Print all list elements and their positions
- Get total sum of numbers in a list
- Get total sum of numbers in a list at even positions
- Get average of numbers in a list
- Find largest number within a list
- Determine is a number exists in a list
- Find the position of a number in a list
- Filter even numbers from a list
- Replace negative numbers with a replacement number

Code available in:

https://github.com/UPRM-CIIC3015-F25/LiveCodingExamples.git

Modifying in place: What happens?

```
def add_one(my_list):
    for x in my_list:
        x = x + 1

A = [1, 2, 3]
add_one(A)
print(A)
```

Modifying in place: What happens?

```
def add_one(my_list):
    for x in my_list:
        x = x + 1

A = [1, 2, 3]
add_one(A)
print(A)
```

Modifying the **loop variable** often results in **unexpected** behavior

To modify in-place, use an accessor pattern

```
def add_one(my_list):
  for i in range(my_list):
    my_list[i] = my_list[i] + 1

A = [1, 2, 3]
add_one(A)
print(A)
```

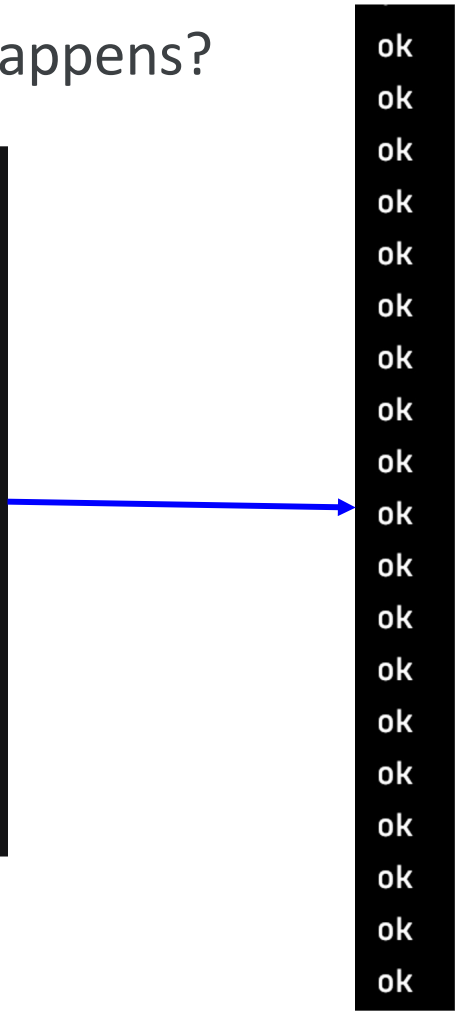
To modify in-place, use an accessor pattern

```
def add_one(my_list):
    for i in range(my_list):
        my_list[i] = my_list[i] + 1

A = [1, 2, 3]
add_one(A)
print(A)
```

Now you are actually updating the list directly, instead of just one variable!

WARNING: Appending in place: What happens?

```
nums = [0]
for _ in nums:
    print("ok")
    nums.append(0)
```



Infinite loop! (Can you see why?)

Review: Lists

What?

Data structure containing **ordered list of elements**, which can be strings, numbers, other data types, and other data structures including lists

It is a **data type**, just like `str`, `int`, `bool`, `float`

Can **index, slice, `in`, iterate**

Why?

Hold data that needs to stay in a certain order

Questions?

Post to our Discord server
or
email: upr-ciic3015-staff@upr.edu

Give us anonymous feedback!

https://bit.ly/CIIC3015_F25_Feedback

Google | GH |

(More about) Strings

Strings Review

- A **string** is a sequence of characters.

```
'wall'   '88 + 99'   'Click here.'   '    '
```

- A **character** is anything typed in a single keystroke, commonly known as a **char**.

```
'r'   '8'   'A'   ','   '&'   ' '   '?'
```

Strings Review: len()

- **len(s)**: returns number of characters in given string

```
len('UPRM')
```
4

```
len(':)')
```
2

```
len('   ')
```
3

```
len('Good morning!')
```
13

Google | GH |

Strings Review: **Concatenation**

Note: Adding strings together is called **concatenation**

\+ Add

```
"And" + "rea" -> "Andrea"
```

* Multiply

```
"hello" * 2 -> "hellohello"

"hello" * 0 -> ""

"hello" * -1 -> ""
```

Note: string multiplication only exists in Python

Google | GH |

Strings Review: **Indexing**

- The **index** is the position of an element in a sequence.
  - Always start counting ***from index 0*** which means last index is always one less than string length: **len(word) - 1**.

- Every **character** in a **string** has an index.
  - For example, the string "Go Tarzans!" looks as follows:

Google

Strings Review: **Indexing**

- In Python, use **square brackets [ ]** to access character at an index

```
username = 'kittens27'
print(username[4])          e
print(username[7])          2
```

Strings Review: **Slicing**

Slice: subsequence of a string

- Start index and end index
- Character at start index is included, character at end index is excluded
- If omit first index (before colon), slice starts at beginning of string
- If omit second index, slice goes to end of string

```
name = 'Colegio'
print(name[1:5])   # 'oleg'
print(name[:7])    # 'Colegio'
print(name[2:])    # 'legio'
```

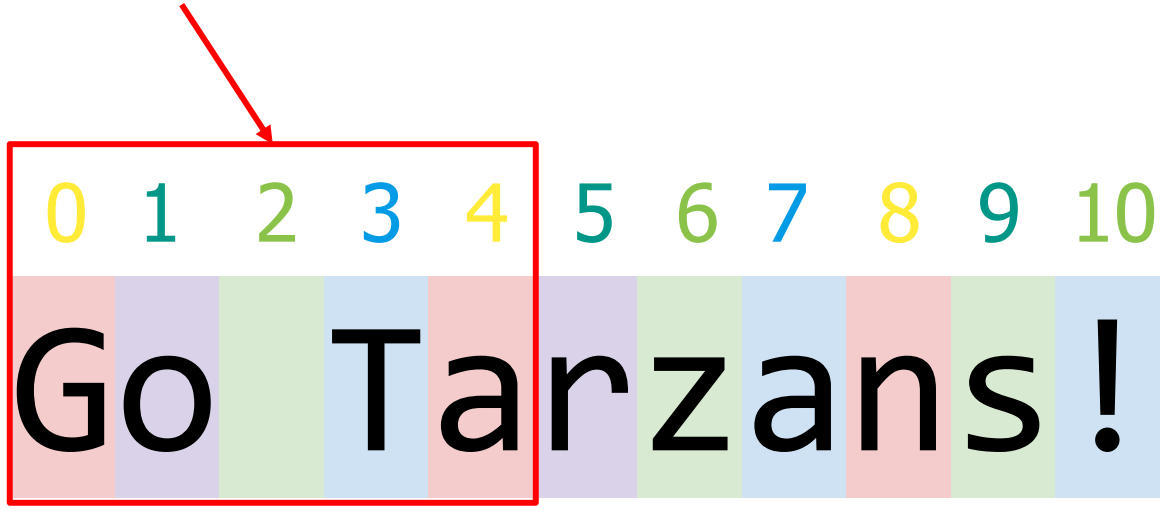
Google | GH |

Strings Review: **Slicing**

Slice: subsequence of a string

Example:

- `string = "Go Tarzans!"`
- `string[0:5] -> "Go Ta"`



Google

Strings Review: **Slicing Practice**

```
name = 'Ariana Grande'

print(name[1:5])   'rian'

print(name[:8])    'Ariana G'

print(name[2:])    'iana Grande'
```

Lists vs. Strings

- **Similarities**:
  - Both are **sequences**
  - Can **index** into, **slice**, and **iterate** over them
  - **in** operator
- **Differences**:
  - Strings can only hold **characters** (strings of length 1)
  - Lists can hold **any data type**, including characters, strings, and other lists

Lists vs. Strings: **Iteration**

```
x = 'qwerty'
for char in x:
 print(char)
```

```
q
w
e
r
t
y
```

```
x = ['qwerty', 'asdf', 'zxcv']
for word in x:
 print(word)
```

```
qwerty
asdf
zxcv
```

Google | GH |

Lists vs. Strings: **Operators**

+ operator concatenates lists together

***** operator repeats a list

```
a = [1, 2, 3]
b = [4, 5, 6]
print(a + b)
print(a)
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3]
```

```
a = [1, 2, 3]
print(a * 2)
```

```
[1, 2, 3, 1, 2, 3]
```

Google | GH |

Lists vs. Strings: **Truthiness**

- **Both can be empty**
  - empty string: `''`
  - empty list: `[]`
- **empty string and list is False-y**
  - `bool([]) == bool('') == False`
- **all non-empty strings and lists are Truthy**
  - `bool(['hello']) == bool('hello') == True`

Live Coding! Example 4
Common String Processing Patterns

- Count vowels in a string
- Find position of first consonant
- Replace all occurrences of a letter by another letter
- Replace first occurrence of a word by another word

Code available in:

https://github.com/UPRM-CIIC3015-F25/LiveCodingExamples.git

Strings: **Built-In Methods!**

- capitalize() Converts the first character to upper case
- lower() Converts a string into lower case
- upper() Converts a string into upper case
- count() Returns the number of times a specified substring occurs in a string
- endswith() Returns true if the string ends with the specified substring
- find() Returns the first position of a substring within a string or -1 if not found
- index() Returns the first position of a substring within a string or error if not found
- isalnum() Returns True if all characters in the string are alphanumeric
- isascii() Returns True if all characters in the string are ascii characters
- join() Joins the elements of a sequence using target string as separator
- replace() Returns a string with all instances of substring replaced with another
- startswith() Returns true if the string starts with the specified substring

AND MANY MANY OTHERS! docs.python.org/3/library/stdtypes.html#string-methods

Strings: **Built-In Methods Examples**

What do you think these do?

```
"test".upper()
"ExAmPlE".lower()
" some string   ".strip()
"destruction".replace("de", "con")
"fun in the sun".replace("un", "ire")
"fun in the sun".replace("un", "ire", 1)
"Hello World".split(sep=" ")
```

“TEST”

“example”

“some string”

“construction”

“fire in the sire”

“fire in the sun”

[“Hello”, “World”]

Google | GH |

Strings: **Escaping**

What happens here if you want to create a string with quotes inside of it?

```
quote = ""Let's get internships" - students"
```

```
quote = '"Let's get internships" - students'
```

Strings: **Escaping**

Some characters have special meaning, like quotes

Sometimes, need "invisible" characters, like newlines

Use escaping!

```
quote = "\"Let's get internships\"\n- students"
print(quote)
```

Strings: **Escaping**

`\n` - newline

`\t` - tab

`\"` - quotation mark

`\'` - apostrophe

`\\` - backslash

Review: Strings

`len(s)`: returns length of s

`s1 in s2`: returns True if `s1` appears in `s2`

`s[i]`: access individual characters in `string` for index `i`

`s[start:end]`: substring from `start` to `end-1`

`s.upper()/s.lower()`: new string with all upper/lower case letters

`s.strip()`: new string from `s` with any leading or trailing spaces removed

`s.replace(s1, s2)`: new string from `s` with `s1` replaced with `s2`

`s.split(sep)` : separates string into a list of words using `sep` as separator

`sep.join(l)` : joins a list of strings into a single string using `sep` as separator

`s.contains(s2)`: returns True of `s2` appears in `s`

`s.index(s2)` : returns position of `s2` in `s`. Throws error if not found

`s.find(s2)` : returns first position of `s2` in `s` and -1 of not found

`s.count(s2)` : returns number of occurrences of `s2` in `s`

Review: Lists

`e in l`: Returns True if element `e` appears in list `l`

`len(l)`: returns length of `l`

`sorted(l)`: returns new sorted version if `l`

`l[i]`: access individual elements in `l` for index `i`

`l[start:end]`: sublist from `start` to `end-1`

`l.index(e)` : returns position of first `e` in `l`. Throws error if not found

`l.count(e)` : returns number of occurrences of `e` in `l`

`l.sort()`: modifies l to be sorted

`l.extend(l2)`: modifies `l` to concatenate `l2`

`l.append(e)`: modifies `l` to add element `e` at the end

Questions?

Post to our Discord server
or
email: upr-ciic3015-staff@upr.edu

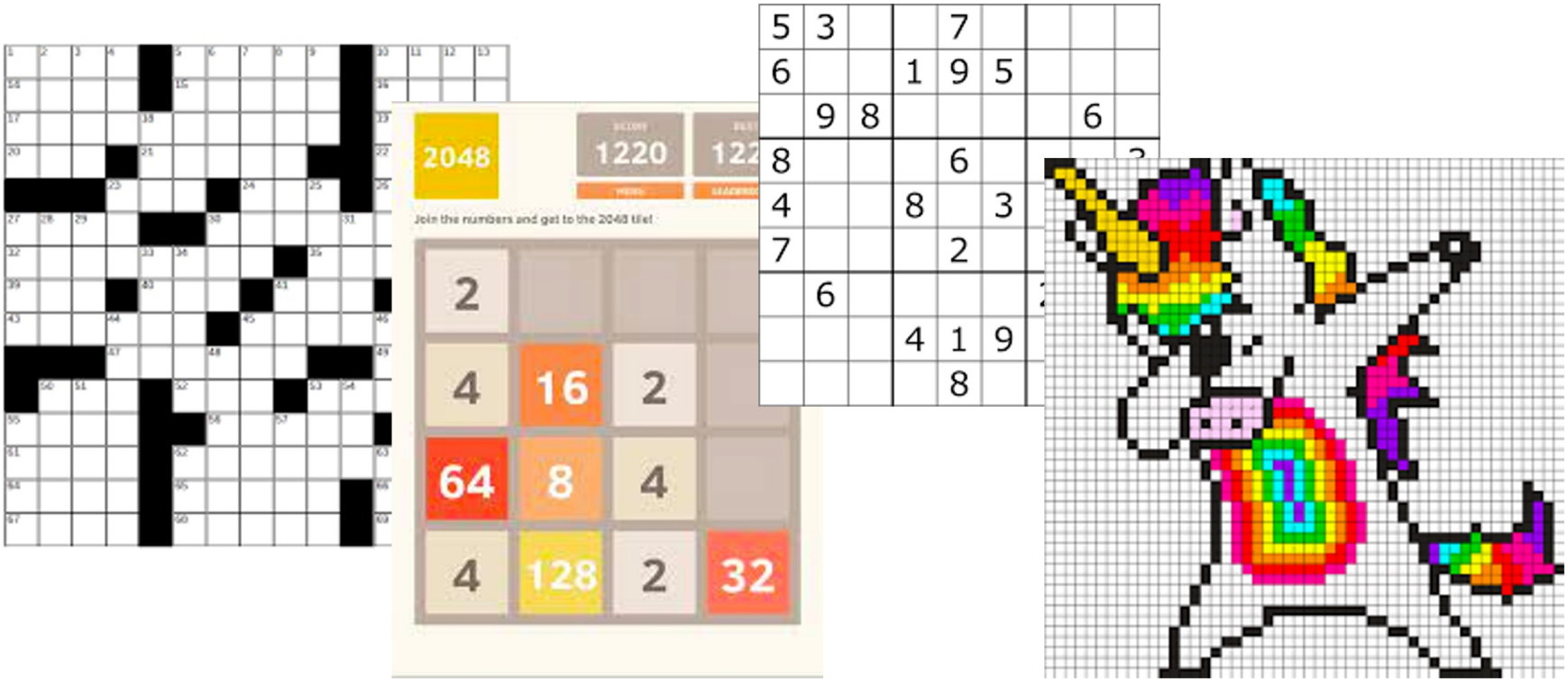
Give us anonymous feedback!

https://bit.ly/CIIC3015_F25_Feedback

Google | GH |

2D Lists and Tables

What can we do with 2D lists?



2D List: **Matrices**

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

1	2	3
4	5	6
7	8	9

2D List: **Candy Crush**

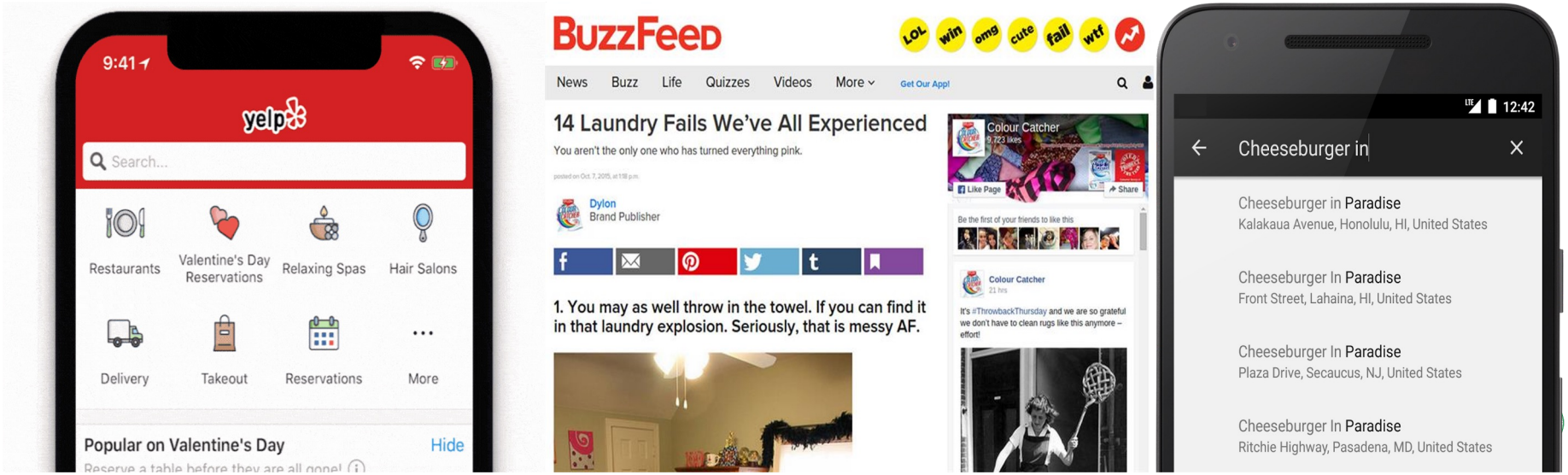
```
[
 ['blue', 'red', 'yellow', 'blue'],
 ['purple', 'fish', 'yellow', 'yellow'],
 ['wall', 'orange', 'red', 'yellow']
]
```

2D (2-dimensional) Lists a.k.a. Tables

- A 2D list is a **list of lists**!
  - Ex: `my_2d_list = [['a', 'b', 'c'], ['a', 'b', 'c']]`
- Can access inner list by indexing
  - Ex: `my_2d_list[0]` -> `['a', 'b', 'c']`
- Can access inner list values by indexing that list
  - Ex: `my_2d_list[0][1]` -> `'b'`

2D List: Databases

- **database**: a collection of organized information to be easily accessed, managed, and/or updated



Internet Movie Database (IMDB)

2D List: Database Records

Song	Artist	Album	Release Year	Genre
Circles	Post Malone	Hollywood's Bleeding	2019	Psychedelic pop
Ransom	Lil Tecca	We Love You Tecca	2019	Hip Hop/Rap
BOP	DaBaby	KIRK	2019	Hip Hop/Rap
Truth Hurts	Lizzo	Cuz I Love You (Deluxe)	2019	Pop
223's (feat. 9lokknine)	YNW Melly, 9lokknine	223's (feat. 9lokknine)	2019	Hip Hop/Rap

2D List: Database Records

one record

Song	Artist	Album	Release Year	Genre
Circles	Post Malone	Hollywood's Bleeding	2019	Psychedelic pop
Ransom	Lil Tecca	We Love You Tecca	2019	Hip Hop/Rap
BOP	DaBaby	KIRK	2019	Hip Hop/Rap
Truth Hurts	Lizzo	Cuz I Love You (Deluxe)	2019	Pop
223's (feat. 9lokknine)	YNW Melly, 9lokknine	223's (feat. 9lokknine)	2019	Hip Hop/Rap

2D List: Database Records

Song	Artist	Album	Release Year	Genre
Circles	Post Malone	Hollywood's Bleeding	2019	Psychedelic pop
Ransom	Lil Tecca	We Love You Tecca	2019	Hip Hop/Rap
BOP	DaBaby	KIRK	2019	Hip Hop/Rap
Truth Hurts	Lizzo	Cuz I Love You (Deluxe)	2019	Pop
223's (feat. 9lokknine)	YNW Melly, 9lokknine	223's (feat. 9lokknine)	2019	Hip Hop/Rap

2D List: **Indexing**

- Just like for lists, can index into 2D lists
- All list functions work for 2D lists, because 2D lists are lists!
- To index: first index “outer” list, then index into resulting “inner” list

2D List: **Indexing**

```
songs = [
0   ['Circles', 'Post Malone', 2019, 'Psychedelic Pop'],
1   ['Ransom', 'Lil Tecca', 2019, 'Hip Hop/Rap'],
2   ['BOP', 'DaBaby', 2019, 'Hip Hop/Rap'],
3   ['Truth Hurts', 'Lizzo', 2019, 'Pop'],
4   ['223\'s (feat. 9lokknine)', 'YNW Melly', 2019, 'Hip Hop/Rap'],
]
```

songs[3]

Google

2D List: **Indexing**

```
                                        songs[3][0]
songs = [
0   ['Circles', 'Post Malone', 2019, 'Psychedelic Pop'],
1   ['Ransom', 'Lil Tecca', 2019, 'Hip Hop/Rap'],
2   ['BOP', 'DaBaby', 2019, 'Hip Hop/Rap'],
3   ['Truth Hurts', 'Lizzo', 2019, 'Pop'],
4   ['223\'s (feat. 9lokknine)', 'YNW Melly', 2019, 'Hip Hop/Rap'],
]
```

Google

2D List: **Indexing**

songs[0]

```
songs = [
0   ['Circles', 'Post Malone', 2019, 'Psychedelic Pop'],
1   ['Ransom', 'Lil Tecca', 2019, 'Hip Hop/Rap'],
2   ['BOP', 'DaBaby', 2019, 'Hip Hop/Rap'],
3   ['Truth Hurts', 'Lizzo', 2019, 'Pop'],
4   ['223\'s (feat. 9lokknine)', 'YNW Melly', 2019, 'Hip Hop/Rap'],
]
```

Google | GH |

2D List: **Indexing**

```
songs[0][3]
songs = [
0   ['Circles', 'Post Malone', 2019, 'Psychedelic Pop'],
1   ['Ransom', 'Lil Tecca', 2019, 'Hip Hop/Rap'],
2   ['BOP', 'DaBaby', 2019, 'Hip Hop/Rap'],
3   ['Truth Hurts', 'Lizzo', 2019, 'Pop'],
4   ['223\'s (feat. 9lokknine)', 'YNW Melly', 2019, 'Hip Hop/Rap'],
]
```

Google

2D List: **Indexing**

songs[4]

```
songs = [
0   ['Circles', 'Post Malone', 2019, 'Psychedelic Pop'],
1   ['Ransom', 'Lil Tecca', 2019, 'Hip Hop/Rap'],
2   ['BOP', 'DaBaby', 2019, 'Hip Hop/Rap'],
3   ['Truth Hurts', 'Lizzo', 2019, 'Pop'],
4   ['223\'s (feat. 9lokknine)', 'YNW Melly', 2019, 'Hip Hop/Rap'],
]
```

Google

2D List: **Indexing**

```
songs[4][2]

songs = [
0   ['Circles', 'Post Malone', 2019, 'Psychedelic Pop'],
1   ['Ransom', 'Lil Tecca', 2019, 'Hip Hop/Rap'],
2   ['BOP', 'DaBaby', 2019, 'Hip Hop/Rap'],
3   ['Truth Hurts', 'Lizzo', 2019, 'Pop'],
4   ['223\'s (feat. 9lokknine)', 'YNW Melly', 2019, 'Hip Hop/Rap'],
]
```

Google

2D List: **Indexing**

```
songs[5]
songs = [
0   ['Circles', 'Post Malone', 2019, 'Psychedelic Pop'],
1   ['Ransom', 'Lil Tecca', 2019, 'Hip Hop/Rap'],
2   ['BOP', 'DaBaby', 2019, 'Hip Hop/Rap'],
3   ['Truth Hurts', 'Lizzo', 2019, 'Pop'],
4   ['223\'s (feat. 9lokknine)', 'YNW Melly', 2019, 'Hip Hop/Rap'],
]
```

IndexError!

2D List: **Indexing**

```
songs = [
0   ['Circles', 'Post Malone', 2019, 'Psychedelic Pop'],
1   ['Ransom', 'Lil Tecca', 2019, 'Hip Hop/Rap'],
2   ['BOP', 'DaBaby', 2019, 'Hip Hop/Rap'],
3   ['Truth Hurts', 'Lizzo', 2019, 'Pop'],
4   ['223\'s (feat. 9lokknine)', 'YNW Melly', 2019, 'Hip Hop/Rap'],
]
```

```
song = songs[1][4]
```

IndexError!

Google

2D List: **Iterating**

```
songs = [
 ['Circles', 'Post Malone', 2019, 'Psychedelic Pop'],
 ['Ransom', 'Lil Tecca', 2019, 'Hip Hop/Rap'],
 ['BOP', 'DaBaby', 2019, 'Hip Hop/Rap'],
 ['Truth Hurts', 'Lizzo', 2019, 'Pop'],
 ['223\'s (feat. 9lokknine)', 'YNW Melly', 2019, 'Hip Hop/Rap'],
]

for song in songs:
  print('Song: ' + song[0])
  print('Artist: ' + song[1])
  print('Year: ' + str(song[2]))
  print('Genre: ' + song[3])
  print()
```

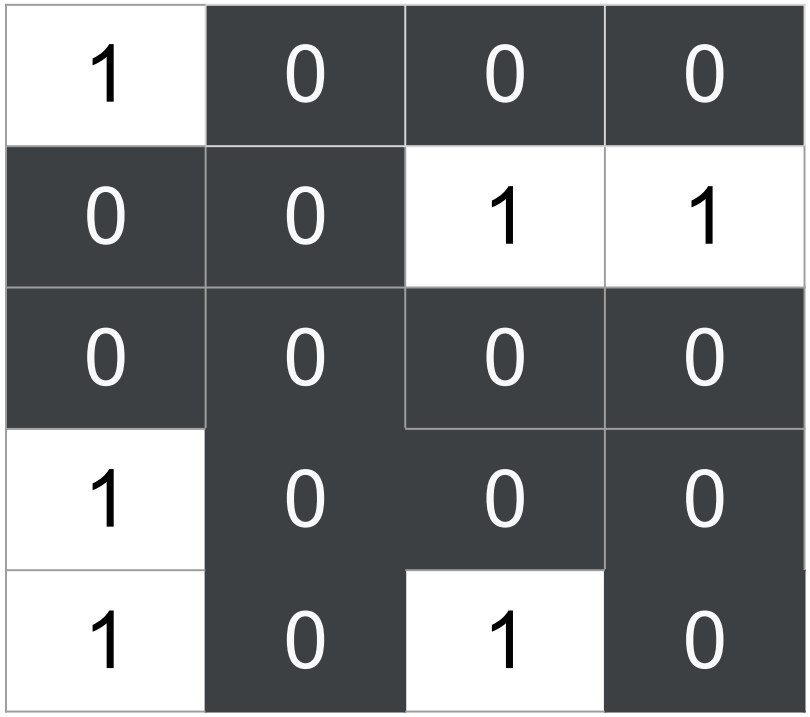
```
Song: Circles
Artist: Post Malone
Year: 2019
Genre: Psychedelic Pop

Song: Ransom
Artist: Lil Tecca
Year: 2019
Genre: Hip Hop/Rap

Song: BOP
Artist: DaBaby
Year: 2019
Genre: Hip Hop/Rap

Song: Truth Hurts
Artist: Lizzo
Year: 2019
Genre: Pop

Song: 223's (feat. 9lokknine)
Artist: YNW Melly
Year: 2019
Genre: Hip Hop/Rap
```

2D List Example: Black and White Images

1	0	0	0
0	0	1	1
0	0	0	0
1	0	0	0
1	0	1	0

Google |

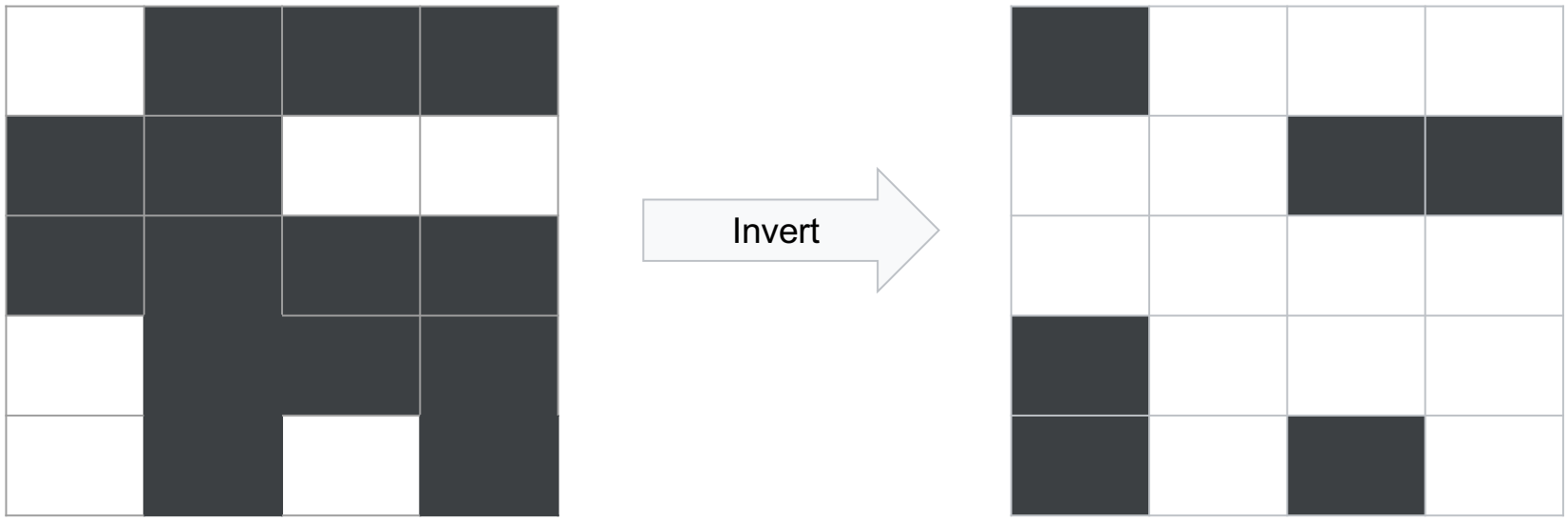
2D List Example: Black and White Images



In Python we represent this image with a nested list

```
A = [
  [1, 0 ,0, 0],
  [0, 0, 1, 1],
  [0, 0, 0, 0],
  [1, 0, 0, 0],
  [1, 0, 1, 0],
]
```

2D List Example: Invert the colors of an image



Example Input

Desired Output

Google | GH |

2D List Example: Invert the colors of an image

```
def invert(my_image):
  for i in range(len(my_image)):
    for j in range(len(my_image[i])):
      if my_image[i][j] == 0:
        my_image[i][j] = 1
      else:
        my_image[i][j] = 0
```

Every element gets mapped to a value.

This is an instance of the map pattern we saw earlier!

2D List Example: Invert the colors of an image

```
def invert(my_image):
  for i in range(len(my_image)):
    print(my_image[i])
    for j in range(len(my_image[i])):
      if my_image[i][j] == 0:
        my_image[i][j] = 1
      else:
        my_image[i][j] = 0
```

What gets printed?

2D List Example: Invert the colors of an image

```
def invert(my_image):  1 usage
 for i in range(len(my_image)):
  print(my_image[i])
  for j in range(len(my_image[i])):
   if my_image[i][j] == 0:
    my_image[i][j] = 1
   else:
    my_image[i][j] = 0
A = [
 [1, 0, 0, 0],
 [0, 0, 1, 1],
 [0, 0, 0, 0],
 [1, 0, 0, 0],
 [1, 0, 1, 0],
]
invert(A)
```

This is printing each row BEFORE the transformation

```
[1, 0, 0, 0]
[0, 0, 1, 1]
[0, 0, 0, 0]
[1, 0, 0, 0]
[1, 0, 1, 0]
```

2D List Example: Invert the colors of an image

```
def invert(my_image):  1 usage
 for i in range(len(my_image)):
  for j in range(len(my_image[i])):
   if my_image[i][j] == 0:
    my_image[i][j] = 1
   else:
    my_image[i][j] = 0
  print(my_image[i])
A = [
 [1, 0, 0, 0],
 [0, 0, 1, 1],
 [0, 0, 0, 0],
 [1, 0, 0, 0],
 [1, 0, 1, 0],
]
invert(A)
```

This is printing each row AFTER the transformation

```
[0, 1, 1, 1]
[1, 1, 0, 0]
[1, 1, 1, 1]
[0, 1, 1, 1]
[0, 1, 0, 1]
```

Live Coding! Examples 5

- Create 2D Lists
- Iterate 2D lists using while and for
- Print each row separately
- Print each item separately
- Calculate the mean of all elements in the 2D List

Code available in:

github.com/CSwithJC/CIIC3015/blob/main/lecture_05/example_07.py

Live Coding! Examples 6

- Count songs in a given genre
- Retrieve the titles and artists of all songs in a given year and genre
- Return the title and artists of the most recent song of a given genre

Code available in:

https://github.com/UPRM-CIIC3015-F25/LiveCodingExamples

Review: 2D Lists

What?

List of lists

Can use all list functions on 2D lists because they are lists

Can **index** and **iterate** through 2D lists to access stored data

Why?

Holds data in particular order in which each item has many elements

databases collection of organized information to be easily accessed, managed, and/or updated

Questions?

Post to our Discord server
or
email: upr-ciic3015-staff@upr.edu

Give us anonymous feedback!

https://bit.ly/CIIC3015_F25_Feedback

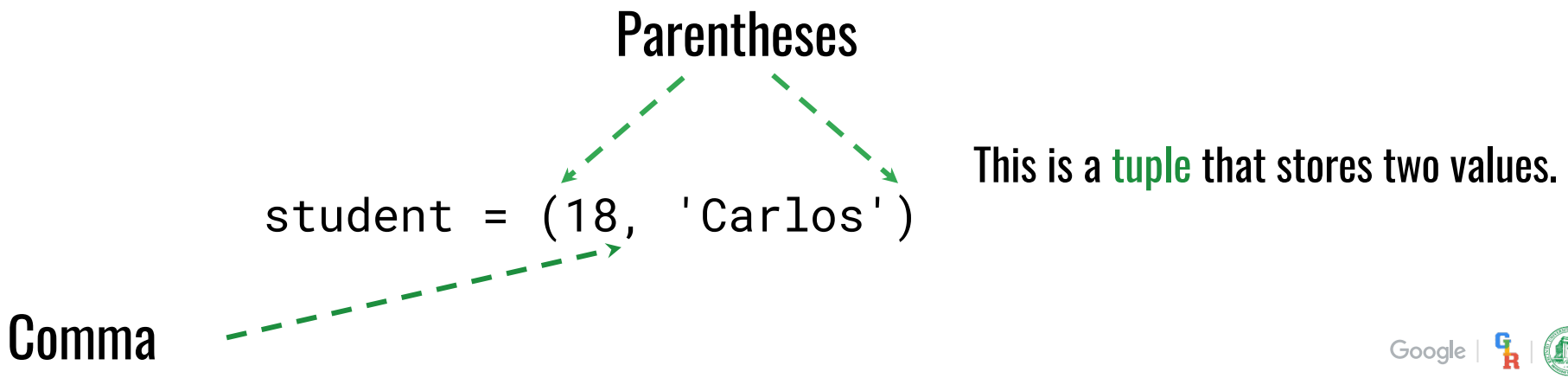
Google | GH |

Tuples

Google | GH |

Tuples

- A tuple is a sequence, very much like a list.
- The primary difference between tuples and lists is that tuples are **immutable**.
  - That means once a tuple is created, it cannot be changed.
- When you create a tuple, you enclose its elements in a set of **parentheses**.



Tuples

- **Tuples** can store any number of values. They are typically used to store multiple values that describe a single item (e.g. student, location, velocity, customer).

```
# tuple with 2 values: age and name
student = (18, 'Carlos')

# tuple with 3 values: lat, lng, location name
location  = (39.346960, -76.582288, 'Calloway')

# tuple with 2 values: x coordinate, y coordinate
velocity = (45, 64)

# tuple with 3 values: id, name, phone number
customer = (1324, 'Amin', '555-5555')
```

Google | GH |

Tuples

- Access values in a **tuple** using the indexing operator.

```
student = (18, 'Carlos')
print(student[0])
print(student[1])
print(student[2])
```

This code prints:

```
> 18
> Carlos
Traceback (most recent call last):
  File "python", line 4, in <module>
IndexError: tuple index out of range
```

Tuples

- Just like strings, **tuples** are **immutable**.

```
student = (18, 'Carlos')
student[0] = 19
```

This code prints:

```
Traceback (most recent call last):
  File "python", line 2, in <module>
TypeError: 'tuple' object does not
support item assignment
```

Tuples vs. Lists

- A **tuple** is a sequence of values of any type
- Tuples are **immutable** meaning you cannot add, remove or replace any values in a tuple
- Best practice is to use tuples to store immutable data that may contain elements of **different** types.

- A **list** is a sequence of values of any type
- Lists are **mutable** meaning you can add, remove or replace values in a list
- Best practice is to use lists to store mutable data with elements of the **same** type

```
PURPLE = (255, 0, 255)
BLUE = (0, 0, 255)
2d_point = (100, 200)
```

Google | GH |

Live Coding! Example 7

- Create Tuples
- Access Elements in a Tuple
- Try to update a Tuple
- Time the creation of a tuple vs. a list using timeit module

Code available in:
github.com/CSwithJC/CIIC3015/blob/main/lecture_05/example_06.py

Questions?

Post to our Discord server
or
email: upr-ciic3015-staff@upr.edu

Give us anonymous feedback!

https://bit.ly/CIIC3015_F25_Feedback

Google | GH |